Accelerating XML Structural Join By Partitioning

Nan Tang[†] Jeffrey Xu Yu[†] Kam-Fai Wong[†] Kevin Lü[‡] Jianxin Li[§]

[†] Department of Systems Engineering & Engineering Management
The Chinese University of Hong Kong, Hong Kong, China
[‡]Brunel University, London, UK
[§]College of Information Science and Engineering, Northeastern University, China
{ntang,yu,kfwong}@se.cuhk.edu.hk, kevin.lu@brunel.ac.uk

Abstract. Structural join is the core part of XML queries and has a significant impact on the performance of XML queries, several classical structural join algorithms have been proposed such as *Stack-tree* join and *XR-Tree* join. In this paper, we consider to answer the problem of structural join by partitioning. We first extend the relationships between nodes to the relationships between partitions in the plane and get some observations. We then propose a new partition-based method *P-Join* for structural join. Based on *P-Join*, moreover, we present an enhanced partitioned-based spatial structural join algorithm PSSJ.

1 Introduction

Structural join can be regarded as the core part of XML queries. A series of structural join algorithms were proposed in the literature. [10] proposed a merge-based join algorithm called *multi-predicate merge join*(MPMGJN). [6] proposed another merge-based join algorithm $\mathcal{EE}/\mathcal{EA}$. [1] proposed *Stack-Tree-Desc/Anc* algorithm which uses stack mechanism to efficiently improve the merge-based structural join algorithm. Then [2] utilized the B^+*-Tree* index on the *Stack-tree* algorithm to overleap the descendant nodes which do not participate in the join. [5] proposed *XR-Tree*, namely, XML Region Tree, to skip both ancestors and descendants nodes that do not participate in the join. In paper [7], an extent-based join algorithm was proposed to evaluate path expressions containing parent-children and ancestor-descendant operations. In order to further improve the query performance, two novel query optimization techniques, path-shortening and path-complementing were proposed in paper [9]. Path-shortening reduces the number of joins by shortening the path while path-complementing optimizes the path execution by using an equivalent complementary path expression to compute the original one.

Many numbering schemes have been utilized in previous algorithms because with them we could quickly determine the positional relationships between tree nodes. Previous strategies mainly focus to the relationship in 1-dimensional space while ignoring the relationships between partitions or subspaces in 2-dimensional space. Grust [4] answered the XPath queries in 2-dimensional space using Dietz numbering scheme. Distinct from structural join, however, [4] only aim at the set operation while structural join gives all the combination of ancestors and descendants, which is obvious more complex and practical.

The major contributions of this paper are summarized as follows:

– We extend the relationships between nodes to the relationships between partitions in a 2- dimensional space and get 9 observations.
– We present a partition-based structural join method, namely, *P-Join.* [2, 5] skip ancestors and descendants nodes based on *Stack-tree* join algorithm. With *P-Join*, however, after filtering operation, the ancestor and descendant nodes are further classified into different areas using positional relationships on plane. Then some portions could directly output join results without actual join operation, other portions could also quickly produce join results utilizing spatial characters.
– We present an enhanced partition-based spatial structural join algorithm PSSJ based on *P-Join.*

The rest of this paper is organized as follows. In Section 2 we introduce the numbering scheme chosen and some concepts about partition relationships. Next, in Section 3 we present a partition-based structural join method *P-Join.* Section 4 presents an enhanced partition-based spatial structural join algorithm. Section 5 gives the performance evaluation. Finally Section 6 concludes this paper.

2 Numbering Scheme and Partition Relationships

There are many proposed numbering schemes to determine the positional relationships between any pair of tree nodes [1, 3, 4, 6]. In this paper, we use the Dietz numbering scheme [3] which uses *preorder* and *postorder* values, as a pair, to encode each tree node. Dietz numbering scheme expresses the positional relationship as follows: (i) For a node e_j and its ancestor e_i, $\textsc{Pre}(e_i) < \textsc{Pre}(e_j)$ and $\textsc{Post}(e_i) > \textsc{Post}(e_j)$. (ii) For two sibling nodes e_i and e_j, if e_i is the predecessor of e_j in preorder traversal, then $\textsc{Pre}(e_i) < \textsc{Pre}(e_j)$ and $\textsc{Post}(e_i) < \textsc{Post}(e_j)$. $\textsc{Pre}(\mathrm{e})$ and $\textsc{Post}(\mathrm{e})$ represent the *preorder* and *postorder* of e, respectively.

The tree nodes relationship can be clearly expressed on a space shown in Figure 1. Consider the node p in Figure 1. Any node e on the top left of p is the ancestor of p because $\textsc{Pre}(e) < \textsc{Pre}(p)$ and $\textsc{Post}(e) > \textsc{Post}(p)$; likewise, the node on the bottom right is the descendant of p; the node on the bottom left is the preceding node of p; the node on the top right is the following node of p.

The main reason of using the Dietz numbering scheme is simply explained below, in comparison with the region code, which uses a *start/end* position of an element to code its position [1]. nodes with the Dietz numbering scheme are distributed dispersedly, whereas the nodes with the region code are rather skewed and only in the upper triangle of a subspace. This dispersed distribution character of Dietz numbering scheme is more appropriate for our spatial-partitioning technique introduced in section 4, thus we choose Dietz numbering scheme. However, our algorithm could also work on region code.

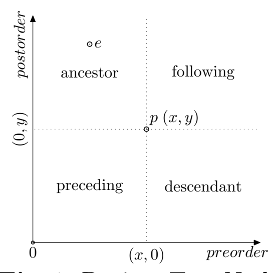


Fig. 1. Project Tree Nodes

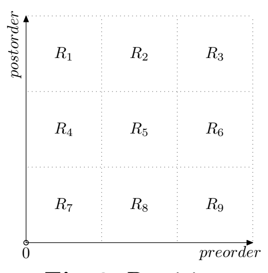


Fig. 2. Partitions

In this paper, we extend the relationships between nodes as shown in Figure 1 to the relationships between partitions. As shown in Figure 2, the whole plane is partitioned into 9 disjoint rectangle partitions, $R_1, R_2, \cdots, R_9$. Here each R_i contains a subset of XML elements (or nodes in an XML tree). Consider the partition R_5 in the center of Figure 2. We made the following observations:

1. All nodes in R_1 are the ancestors of all nodes in R_5.
2. All nodes in R_3 are the "following" nodes of all nodes in R_5.
3. All nodes in R_7 are the "preceding" nodes of all nodes in R_5.
4. All nodes in R_9 are descendants of all nodes in R_5.
5. Some nodes in R_2 are ancestors or "following" nodes of some nodes in R_5.
6. Some nodes in R_4 are ancestors or "preceding" nodes of some nodes in R_5.
7. Some nodes in R_6 are descendants or "following" nodes of some nodes in R_5.
8. Some nodes in R_8 are descendants or "preceding" of some nodes in R_5.
9. Some nodes in R_5 may have any positional relationships with nodes in R_5.

The observation made above shows that we can process structural join using a partition-based approach. When users query the ancestors of nodes in R_5, i) we do not need to do structural join between R_5 and any of R_3, R_6, R_7, R_8 and R_9, because none of these partitions includes ancestors of any nodes in R_5; ii) we do not need to do structural join between R_1 and R_5, because all nodes in R_1 are the ancestors of all nodes in R_5; and iii) we only need to do structural join between R_2, R_4, R_5 and R_5 because some of nodes in R_2, R_4 and R_5 are the ancestors of some nodes in R_5. The similar techniques can be applied when users query the descendant or other XPath axes in the partition of R_5.

3 A New Partition-Based Structural Join Method

In this section, we propose a new partition-based method for structural join between ancestors and descendants. Our method can be easily extended to support other XPath axes including following, preceding, sibling, etc.

First, we show several cases in Figure 3, and then we outline our partition-based structural join method. Figure 3 (a) shows a general case where a 2-dimensional space is partitioned into 4 partitions, A_1,A_2, A_3 and A_4 according to D. Here, D is a minimum partition (rectangle) that contains all descendants. Note that D is always located the same as A_4 for any structural join based on ancestor/descendant relationships.

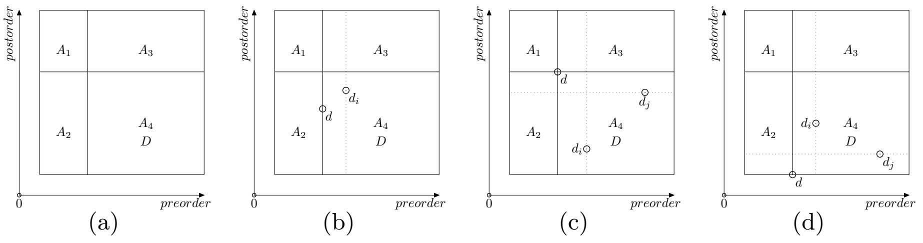


Fig. 3. Several Cases

One technique we use in our partition-based join is to dynamically adjust the partitions of A_1, A_2, A_3 and A_4 $(= D)$ when we attempt to find ancestors for every $d \in D$. In other words, the four partitions are conceptually redrawn for any $d \in D$. Two ascending B^+-*Tree* indices on the *preorder* and *postorder* value are created for A, respectively, to guarantee the efficent search operation. We only need to scan every $d \in D$ once. The details are given below. Assume that elements in partitions are sorted by their *preorder* values in ascending order. The adjustment of partitions has the following three cases for a node $d \in D$ with minimum *preorder* value. i) d is in the middle of the boundary between A_2 and D (Figure 3 (b)). ii) d is at the top of the boundary of A_2 and D (Figure 3 (c)). iii) d is at the bottom of the boundary between A_2 and D (Figure 3 (d)).

For the first case, we output d with all a in A_1, then traverse each a_i in A_2 in the reverse order and compare it with d, if a_i is not the ancestor of d, then all elements d_i after d cannot be the descendant of a_i, we do not need to join this a_i with any such $d_i \in D$. Otherwise, if a_i is an ancestor of d, all a_j before a_i are ancestors of d. Suppose, after processing d, we need to process the next descendant d_i in the sorted list (Figure 3 (b)). Because d is on the boundary between A_2 and D, the boundary between A_3 and $d_i \in D$ will remain unchanged. We can find those elements $a \in A_3$ such as $\textsc{Pre}(a) < \textsc{Pre}(d_i)$. Note when processing $d_i \in D$ in Figure 3 (b), those nodes a in A_4 such as $\textsc{Pre}(a) < \textsc{Pre}(d_i)$ are adjusted to A_2. It is important to note that the area of A_1 and A_2 are expanded while the area of A_3 and A_4 are shrunk. This adjustment process is shown in Figure 3 (b).

The second case is simple, because only all nodes in A_1 are the ancestors of d. All nodes in A_2 are the preceding nodes of d, all nodes in A_3 are the following nodes of d, and all nodes in A_4 are the descendants of d. We can simply output d with every $a \in A_1$ as join results. After processing d, we need to adjust the boundaries of D, when we process the next descendant d_i which has the smallest *preorder* value. In the first step, we identify a node $d_j \in D$ with the largest *postorder* value from the unprocessed descendants. In the second step, we adjust the following nodes a to the A_1: 1) $a \in A_2$ if $\textsc{Pre}(a) > \textsc{Post}(d_j)$, 2) $a \in A_4$ if $\textsc{Pre}(a) > \textsc{Post}(d_j)$, 3) $a \in A_3$ if $\textsc{Pre}(a) < \textsc{Pre}(d_i)$, and 4) $a \in A_4$ if $\textsc{Pre}(a) < \textsc{Pre}(d_i)$.

And in the third step of above case, we adjust those nodes $a \in A_4$ satisfying $\textsc{Post}\,(a) > post(d_j)$ to A_3; and $a \in A_4$ satisfying $post(a) < post(d_j)\&pre(a) < pre(d_i)$ to A_2. This adjustment process is shown in Figure 3 (c).

For the third case, all nodes in A_1 and A_2 are ancestors of d, and no nodes in A_3 and A_4 are ancestors of d. After processing d, we get the next element $d_i \in D$ following the sorted order. We can determine a node d_j which has the minimum *postorder* value from the remaining elements. Because d is at the bottom of the boundary between A_2 and D, the boundaries need to be adjusted. Here, we first remove those the elements a in A_2 and A_4 satisfying $\textsc{Post}\,(a) < \textsc{Post}\,(d_j)$, because these nodes a will not be ancestors of any descendants $d \in D$. Second, we adjust those elements $a \in A_3$ satisfying $\textsc{Pre}\,(a) < \textsc{Pre}\,(d_i)$ to A_1. Third, we adjust those elements $a \in A_4$ satisfying $\textsc{Pre}\,(a) < \textsc{Pre}\,(d_i)$ to A_2. This adjustment process is shown in Figure 3 (d). Above method is our partition-based structural join method, P-Join.

The main advantage of method *P-Join* is that it does not necessarily join every ancestors and descendants using a dynamic boundary adjustment technique. In fact, as a pre-processing technique, we can filter those nodes that cannot be matched before calling *P-Join*. Filter operation is extensively used in database system to save I/O cost, consequently improve the system performance. Traditional bit filter techniques in relational database utilize the equal relationship between attributes of two relations for filtering. For semistructured data like XML, however, bit filter techniques cannot be applied well, thus we filter nodes with their spatial positional characters.

Now we explain the filtering operation. First, we use the minimum rectangle containing all descendants d to filter A, all ancestors a in the area that cannot be the ancestor of any d are filtered. And then we use the minimum rectangle containing the remaining A to filter D, all nodes d in the area that will not be a descendant of any a will be filtered. It is easy to see that the filtering order of A and D is arbitrary and the results are the same. Moreover, one mutually filtering of A and D is enough and there is no need to recursively filter.

4 An Enhanced Partition-based Spatial Structural Join Algorithm

In this section, we further investigate on how to partition A and D when querying the ancestor-descendant relationship $a /\!/ d$. We will present a spatial partitioning approach. On top of the spatial partitioning approach, an enhanced partition-based spatial structural join algorithms is proposed.

4.1 Spatial-Partitioning

Partitioning A and D for minimization of structural join is challenging, and is itself complex and time consuming. To find an efficient partitioning method, we are now left with the challenge to find a method which considers the spatial characters of A or D first, and then partition the left D or A for structural join.

When the size of D is large, the nested level is high and the distribution is bad-proportioned, the partitioning methods that mainly consider D will get good performance. On the contrary, the methods which pay attention to A will be better. The partitioning order of A or D is symmetrical. In this section, we mainly discuss the partitioning method for D, the method for A can be applied in a similar manner.

Logically, an element node should be potentially distributed in any area of the 2-dimensional space. In practice, nevertheless, for a general XML data tree, element nodes are concentratively distributed near the area from the bottom left to top right of the 2-dimensional space. The root node will exist on the top left. Only when the whole XML data tree is a linear structure, the node will appear on the bottom right, which is obviously an abnormal case. Based on the characteristics of the general distribution of XML data trees, we propose an efficient spatial partitioning method, denoted SPATIAL-PARTITIONING.

The basic idea of SPATIAL-PARTITIONING is first to find two data marks that have the largest distance in D and search the nearest $\frac{|D|}{N}$ nodes of each data mark with a NEAREST method. Spatial-Partitioning will recursively call the NEAREST method until the size of a group is 1 or 2. The two data marks are chosen from the edges of the minimum rectangle containing all d. The selection of the $\frac{|D|}{N}$ nearest nodes is implemented as shown in Figure 4. The NEAREST() procedure is given below:

```
Nearest($x, n$)
 1  $nX \leftarrow x$ 's nearest  $n$  points in x-axis
 2  $e \leftarrow$ FURTHERMOST($x, nX$)
 3  $r \leftarrow$ DISTANCE($e, x$)
 4  $p \leftarrow$ $minPreorder$ node in the left nodes
 5  while X-DISTANCE($p, x$) $< r$
 6      do if DISTANCE($p, x$) $< r$
 7            then $nX \leftarrow nX - e + p$
 8                 $e \leftarrow$ FURTHERMOST($x, nX$)
 9                 $r \leftarrow$ DISTANCE($e, x$)
10                 $p \leftarrow p.next$
11  return $nX$
```

In the NEAREST procedure, the FURTHERMOST(x, nX) searches the furthermost point of x in nX, the DISTANCE(e, x) is the $2D$ distance between e and x, the X-DISTANCE(p, x) is their horizontal distance.

We give an example for searching five nearest nodes of x including x in Figure 4. We first search five nearest nodes on the X-axis distance of x, which are x, e_1, e_2, e_3 and e_4. The furthermost node is e_1, and r_1 is the distance between x and e_1, any node whose horizontal distance with x is less than r_1 may be in the result. We choose node according to its preorder, so next we get e_4, which is nearer than e_1, so we remove e_1 and add e_4. Now the five nodes are x, e_2, e_3, e_4 and e_5. And then, we get e_6 and remove e_3, now the furthermost node is e_4

and the furthermost distance is r_3, other nodes' horizontal distance with x are all larger than r_3. Now we get $x's$ five nearest nodes x, e_2, e_4, e_5 and e_6.

Compared with computing all nodes to find their nearest n nodes, this partitioning method only need compute a small part of all nodes to get the results. Furthermore, along with the progress of partitioning, the nodes will be less and the search speed will be quicker. The efficiency of NEAREST procedure profits from the particular plane distribution characters of tree nodes. With above example, we can easily see that our partitioning method could quickly partition D into N groups, each has $\frac{|D|}{N}$ nodes. We show SPATIAL-PARTITIONING algorithm below:

```
Spatial-Partitioning($D, N, i = 1$)
switch
    case $N = 1$ :
        return $D_i$
    case $N = 2$ :
        choose $p_1$ and $p_2$ in $D$
        $D_i \leftarrow$ NEAREST($p_1, |D|/N$);    $D_{i+1} \leftarrow$ NEAREST($p_2, |D|/N$)
        return $D_i, D_i + 1$
    case $N > 2$ :
        choose $p_1$ and $p_2$ in $D$
        $D_i \leftarrow$ NEAREST($p_1, |D|/N$);    $D_{i+1} \leftarrow$ NEAREST($p_2, |D|/N$)
        $D \leftarrow D - D_i - D_{i+1}$;    $N \leftarrow N - 2$;    $i \leftarrow i + 2$
        GROUP($D, N, i$)
        return $D_1 \cdots D_N$
```

In our SPATIAL-PARTITIONING() method, if $N = 1$, we just distribute all D to one group. Otherwise, choose two enough remote datum marks on the edges of minimum rectangle containing all D, and search each datum mark's $\frac{|D|}{N}$ nearest points and distribute them to one group. And then, we recursively call the spatial partitioning method until all D are distributed to N groups. With this procedure, the number of d in each group is $\frac{|D|}{N}$.

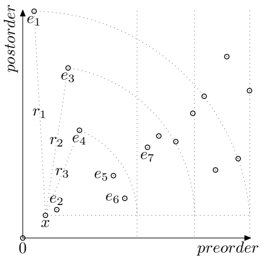


Fig. 4. Search n Nearest Nodes

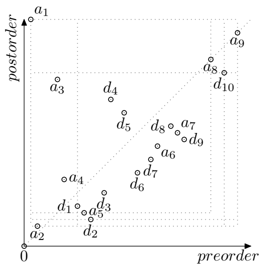


Fig. 5. Node Distribution

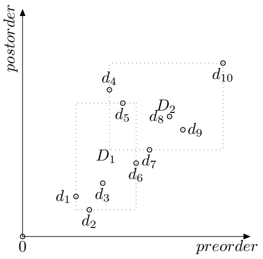


Fig. 6. Partition D

To make the result simple and concrete, suppose partitioning number $N = 2$. We partition the D as shown in Figure 5, we first choose two enough remote points d_1 and d_{10}. Then choose d_1's $\frac{|D|}{N} = 5$ nearest points d_1, d_2, d_3, d_5, d_6 to first group D_1 and $d_4, d_7, d_8, d_9, d_{10}$ to the second group D_2. The partitioning result is shown in Figure 6. D_1 is the minimum rectangle containing all d nodes inside D_1, and D_2 is the minimum rectangle containing all d nodes inside D_2.

4.2 Partition-baesd Spatial Structural Join Algorithm

Based on the Spatial-Partitioning method, we propose an enhanced partition-based spatial structural join algorithm, *PSSJ*.

PSSJ algorithm first filter most ancestors and descendants which will not be join results. Then use basic Spatial-Partitioning method to partition D. For each group in the spatial partition, we call *P-Join* method.

5 Performance Analysis

In this section, we present the experimental evaluation that yields a sense for the efficacy of our novel partition-based spatial structural join algorithm.

5.1 Experiment Setting

We ran experiments on a PC with Intel Pentium 2.0 GHz CPU, 512M RAM and 40G hard disk. The operating system is Windows XP. We employ an object-oriented XML management system XBase [8] for storage. The testing programs were written in INADA conformed to ODMG *C++* binding. We have implemented PSSJ, *Stack-tree* join (STJ) and *XR-Tree* join algorithms.

We use synthetic data for all our experiments in order to control the structural and consequent join characteristics of the XML data. We use the IBM XML data generator to generate XML documents. The two DTDs adopted are department DTD and library DTD. Different sized documents that scale from 20M to 100M are generated with each DTD.

Experiments were performed to study the performance on our partition-based spatial structural join algorithm. Table 1 shows the set of queries for all data set on two DTDs. It also presents the nested case of our query set. The test query set contains all possible nested case of ancestor and descendant for thoroughly analyzing the performance of partition-based spatial structural join algorithm.

Query	*Description*	*DTD*	*Anc Nested*	*Desc Nested*
Q_1	department//employee	Department	highly	highly
Q_2	employee//name	Department	highly	less
Q_3	section//subsection	Library	less	highly
Q_4	section//title	Library	less	less

Table 1. Query Set and Nested Case

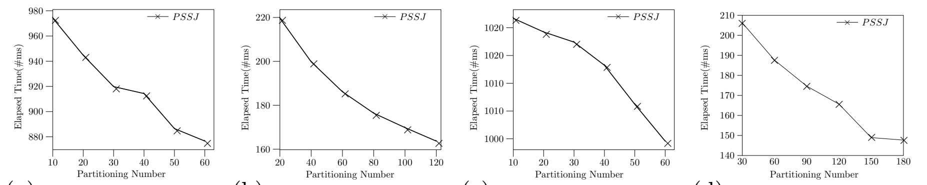
5.2 Varying Partitioning Number

The objective of this set of experiments is to test the query performance of PSSJ with varying partitioning number. We adopt one 100M document for each

DTD. The performance curves of testing varying partitioning number are given in Figure 7.

Figure 7(a) shows the performance of Q_1 with partitioning number varying from 10 to 60. Figure 7(b) shows the performance of Q_2 with partitioning number from 10 to 60. Figure 7(c) shows the performance of Q_3 with partitioning number from 20 to 120. Figure 7(d) shows the performance of Q_4 with partitioning number from 30 to 180.

It can be seen from Figure 7 that PSSJ algorithm can get better performance along with the increase of the partitioning number, which demonstrates the effectiveness of our partitioning methods. We can also see that when the partitioning number gets to some value in some cases, the amplitude of performance will be slow. This value is closely relative to the number and the nested case of ancestors and descendants.

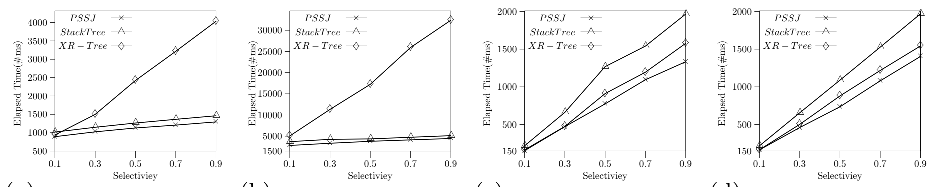


(a) Q_1(department vs. employee) (b) Q_2(employee vs. name) (c) Q_3(section vs. subsection) (d) Q_4(subsection vs. title)

Fig. 7. Varying Partitioning Number Test

5.3 Varying Join Selectivity Test

In the second group of experiments, we study the capabilities of various algorithms to skip elements under different join selectivity. We keep the document size 100M and partitioning number unchanged for each query in this test. We test the case when the ancestor join selectivity and descendant join selectivity are varying. For this purpose, we change the document elements with dummy elements so that the desired selectivity on ancestors or descendants can be obtained. Figure 8 shows the performance of the three algorithms tested when varying join selectivity.



(a) Q_1(department vs. employee) (b) Q_2(employee vs. name) (c) Q_3(section vs. subsection) (d) Q_4(subsection vs. title)

Fig. 8. Varying Join Selectivity

Figure 8(a) tests Q_1 and Figure 8(b) tests Q_2. The nested level of ancestors are high in both queries. We can clearly see from them that in this case, the join selectivity has great influence on *XR-Tree* algorithm, while has little influence on Stack-Tree join algorithm and our partition-based spatial join algorithm. And in this case, *XR-Tree* has bad performance. Stack-Tree join algorithm has similar good performance as PSSJ.

Figure 8(c) tests Q_3 and (d) tests Q_4. The nested level of ancestors is low. We can see from (c)(d) that join selectivity has great influence on all tested algorithms. And in this case, *XR-Tree* is better than Stack-Tree join and a little worse than PSSJ.

6 Conclusions

In this paper, we extend the relationships between nodes to the relationships between partitions and get some observations about the relationships. We then propose a new partition-based structural join method *P-Join* for structural join between ancestors and descendants based on the observations. Moreover we present an enhanced partitioned-based spatial structural join algorithm PSSJ. Extensive experiments show the excellence of our new approach.

Acknowledgement

This work is partially supported by the CUHK strategic grant(numbered 4410001).

References

1. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE '02*, 2002.
2. S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of VLDB '02*, 2002.
3. P. F. Dietz. Maintaining order in a linked list. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, 1982.
4. T. Grust. Accelerating XPath location steps. In *Proceedings of SIGMOD '02*, 2002.
5. H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient stuctural join. In *Proceedings of ICDE '03*, 2003.
6. Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *Proceedings of VLDB '01*, 2001.
7. G. Wang and M. Liu. Query processing and optimization for regular path expressions. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, 2003.
8. G. Wang, H. Lu, G. Yu, and Y. Bao. managing very large document collections using semantics. *Journal of Comput. Sci. & Technol.*, 18(3):403–406, Jule, 2003.
9. G. Wang, B. Sun, J. Lv, and G. Yu. Rpe query processing and optimization techniques for xml databases. *Journal of Comput. Sci. & Technol.*, 19(2):224–237, March, 2004.
10. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD '01*, 2001.